Functional Programming

What is functional programming?

1. Every function must take an argument: f(x)
2. Every function must return a value: y = f(x)
3. Calling a function f(x) with the same input x always returns the same output y
4. Functions just like any other data and can be stored in variables and passed as arguments
5. Functions are pure and have no side effects
6. Side effects: a change to state outside a function which causes it to behave differently each time it runs.
7. All variables are “immutable” and can never be modified! `F(x) { x = x + 1; return x; }`

Definition of Pure Function:

A function that has two properties: Given a specific input x, the function always returns the same output y. It computes its output exclusively based on its input parameters, without relying on or modifying external data that might change from call to call. Ex: Pure

```
int f(int p) {
    int q = 5*p*p
    return q;
```

Ex: Impure

```
int z
int f(int p) {
    return p*z++;
}
```

- code is easier to reason about, debug, test
- enables parallelism, compiler optimizations
- formally analyze, prove things about programs

Imperative vs. Functional Programming

Imperative

- algorithmic - series of statemetns and variable changes
- changes in variable state are required
- sequences of statements, loops, and function calls.
- multi-threading is tricky and prone to bugs
- order code executes is important

Functional Programming

- Transformation through function calls.
- All "variables" are constants - no changes are allowed!
- Function calls and recursion - no loops, no statements!
- mutli-threading is easy!
- Order of execution is of low importance

- In imperative coding languages like C++, the compiler has to strictly call functions in the order that they are written. Functions have side effects and changing the order that they're called can yield totally different results.
- Meanwhile in Functional Programming Languages, functions don't have side effects and thus for any given input, a function will always produce the same output no matter when it's called! Because of this predictability in behavior, Haskell can use lazy evaluation only when it is forced to actually evaluate something.

REPL: Interpreter Reads commands from the user, Evaluates them, Prints the results, then Loops (waiting for the next command)

Haskell DataTypes

- Statically Typed, utilizes type inference
- Type of all variables and functions can be decided at compile time Dynamically Typed
- All types are determined at run time
- Type signatures in Haskell are entirely optional

Semantics: / can only be used for floats and doubles take n = a function that returns a new list that contains the first n items of the old list drop n = a function that returns a new list without its first n items. !! n = a function that returns us the nth item in the list elem = a function that tests whether the specified item was found in the list, returns True if it is sum = a function that adds up every item in a list of numbers product = a function that multiplies up every item in a list of numbers or = takes the logical or of all booleans in a list and = takes the logical and of all booleans in a list zip = creates a new list with tuples containing items from the first and second lists

Composite DataTypes

Tuples, List, Strings

Tuples: allows us to group related information, fields together

Lists: can hold zero or more items of the same type

- Haskell lists are implemented as Linked Lists and have are big O(N) time in accessing that nth item of the list
- **String Theory**
  String is syntactic sugar for a list of chars

head takes the first value of a list tail takes the remainder of the list head and tail work on strings because Strings are ultimately lists of characters

Constructing Lists: Concatenating & Consing

++ operator takes two lists and returns a new list that's a concatentation of them

: operator concatenates the left value to the front of a list, and returns a new list

Range in haskell using the .. in a list can continue a list onto a specific range based on the existing values in the list Comprehensions: List comprehensions are used to generate arbitrary complex lists of items with a super-compact "declarative" syntax

- Let's you easily create a new list based on one or more existing lists.
- Inputs:

- ◦ specify one of more input lists that you want to draw from (generators)
- a set of filters on the input lists ("guards")
- a transformation applied to the inputs before items are added to the output list
- [function performed on it | input list range, guard]

Ex:

```
prods = [x * y | x <- [3, 7], y <- [2, 4, 6]]
for i in x:
    for j in y:
```

Haskell Functions:

1. Type information about the function's parameters and return value
2. The function's name and parameter names
3. An expression that defines the function's behavior

Type signature is always optional in haskell as it can infer the parameter and return types based on the function's code.

length is a built in function that will return the value of a list fromIntegral converts an Int to a Double

any_type is a type variable: Haskell's equivalent of generic type, let's you pass whatever type you want (e.g., Int, String, etc.)

Haskell evaluates function calls from left to right: function application is "left associative" ((f g) x) != f(g(x)) f (g x) = f(g(x)) Functions are always called before evaluating operators in haskell f(x) = f x f(x, y) = f x y f(g(x)) = f (g x) f(x, g(y)) = f x (g y) f(x)g(y) = f x * g y

"let" and "where" are constructs that allow you to bind/associate one or more temporary variables with expression(s) for use in your function.

"where" goes at hte end of a function and allows us to bind the variable in scope

When defining a variable for use across multiple expressions, use where * for guards and case expressions

Control Flow in Functions

Every if excpression must have an else clause because every haskell expression must yield a value, omission would mean that there's a chance it wouldn't return anything

We can also do guard syntax:

if then else Can also be written as: | =

Ex:

```
somefunc param1 param2
    | <if-x-is-true> = <run-this>
    | <else-if-y-is-true> = <run-that>
    |  otherwise = <run-this-otherwise>
```

Pattern Matching Simplifies writing functions that process tuples and lists Define multiple versions of the same function same number and types of arguments

Each version of the function may specify required value(s) for one or more of its arguments Can use an underscore if we want to ignore a field in the tuple

List patterns

- can replace formal parameter of a function with a list pattern
- a list pattern is enclosed in () and has two or more variable names separated by colons Ex:

```
(first:rest_of_list)
(first:second:rest_of_list)
```

- the first n-1 variables refer to the first n-1 values in the list passed into the function
- last variable referes to the full tail of the list

(x:[]) <- means that the "rest" of the list must be empty and only runs if the list has just one item

First Class Functions

First Class Function: a function is treated like any other data, they can be stored in variables, passed as arguments to other functions, returned as values by functions, and stored in data structures

First-class Functions enable more efficient program decomposition and enable functions to generate new functions from scratch

First-class functions go hand-in-hand with higher-order functions higher-order functions: a function that accepts another function as an argument

- a function that returns another function as its return value

Passing Functions As Arguments

first-class functions can be passed as arguments

Returning Functions from Other Functions

Another aspect of first-class functions is being able to return them (and store them in variables)

First-Class and higher-order functions are a pillar of virtually all modern programming languages

- Allows us to provide comparison functions for sorting
- Allows us to perform callbacks when events trigger
- Enables multithreading

A Fundamental set of Higher-Order Functions

Mappers, Filters, Reducers Mappers: performs a one-to-one transformation from one list of values to another list of values using a transform function. Filters: filters out items from one list of values using a predicate function to produce another list of values Reducers: operates on a list of values and collapses them into a single output value

Higher-Order Functions: map A mapper function maps a list of values to another list of values. Haskell provides a mapper function called "map" that accepts two parameters:

1. A function to apply to every element of a list.

2. A list to operate on. The type signature of map is map :: (a -> b) -> [a] -> [b]

The first argument to map is a function that converts an individual item from a value of type a to a value of type b. (note: a and b can be the same) The second argument is a list of type a. The map function returns a list of type b as its result. Map Function Code:

```
map :: (a -> b) -> [a] -> [b]
map func [] = []
map func (x:xs) =
    (func x) : map func xs
```

Higher-Order Functions: filter A filter - a function that filters items from an input list to produce a new output list. filter accepts two parameters

1. A function that determines if an item in the input list should be included in the output list.
2. A list to operate on Type Signature of filter is :: filter :: (a -> Bool) -> [a] -> [a]

The first argument to filter is a predicate function that determines if a vlue from the input list should be included in the output list. If the function returns True for a value, then we keep the value in the output list. Otherwise we filter it. The second argument is a list of type a. It returns all items that passed the filter in its output list.

Higher-Order Functions: reducers A reducer is a function that combines the values in an input list to produce a single output value. Each reducer takes three inputs:

1. A function that processes each of the elements
2. An initial "accumulator" value
3. A list of items to operate on.

Haskel has two different reducer functions: foldl and foldr

foldl Pseudocode: foldl(f, initial_accum, lst): accum = initial_accum for each x in lst: accum = f(accum, x) return accum

- we have a function that must accept two arguments, initial_accum which is typically 0, 1 or an empty list, then we have a list of items we want to process Each new value is "folded" into the accumulator as it's processed

  ```
  foldl f accum [] = accum
  foldl f accum (x:xs) =
    foldl f new_accum xs
    where new_accum = (f accum x)
  ```

- If the input list is empty we return the accumulator value, otherwise we break out list into the first item x, and all remaining items xs.
- We then call f on our existing accumulator value and x to compute the new accumulator
- Here the function is calculated first in the new_accum before it is called again in the recursive call, thus the accumulator is calculated first and if there was infinite recursion it wouldn't fail.

foldr Pseudocode:

```
foldr(f, initial_accum, lst):
    accum = initial_accum
    for each x in lst from back to front:
        accum = f(x, accum)
    return accum
```

- Traverse from xn-1 to x0
- The accumulator is passed in second, and x is passed in first.
- This is the opposite of how we pass calues to f in foldl and has implications for non-associative operators (like / or -)

```
foldr f accum [] = accum
foldr f accum (x:xs) =
  f x (foldr f accum xs)
```

```
- If the input list is empty we return the accumulator value,
- First we process the remainder of the list recursively and get the result
- Then we call our function f passing in the current x value, and the result from the rest of the list
- So here the differenc is that we calculate the recursively, we have to wait for the foldlr recursion to finish to then propogate the result back up the call-stack.
-
Advantage is that we can call map reduce in parallel

Go
Go has native support for concurrent execution.
No formal classes or inheritance!
A single, prescribed standard for code formatting

Advanced topics in Functional Programming:
Lambda Functions, Partial Application, Currying, Algebraic Data Types, Immutable Data Structures

Lambda Functions:
just like any other function, but it doesn't have a function name!
Normal Function:
```

```
cube x = x^3 \x -> x^3
```

```
We use "lambdas" in higher-order functions when we don't want to bother defining a whole new named function


How to define lambda functions in haskell:
\param1 ... param2 -> expression
When you specify the names of one or more parameters, separated by spaces. These are called "bound" variables.
-> is used to indicate the expression is next

Thus, we define our function's expression (what the function computes and returns)
```

Lambdas and Closures - Capturing Variables
Ex:
slopeIntercept m b = (\x -> m*x + b)
twoxPlusOne = slopeIntercept 2 1
This function builds and returns a new function that takes one argument x and computes y = m*x + b

Thus, Haskell captures these specific values m and b along with our expression

Another term for a variable that is captured as part of a closure is a free variable.
Free variable - any variable that is not an explicit parameter to the lambda
In our example above, x is not a free variable since its value is passed in to the lambda

Closure
A combination of two things:
1. A function of zero or more arguments that we wish to run at some point in the future.
2. A list of "free" variables and their values that were captured at the time the closure was created

When a closure later runs, it uses the values of the free variables captured at the time it was created*

Closures are implemented as a struct of a function, a list, and map of variables

Partial Function Application:
an operation where we define a new function g by combining:
1. An existing function f that takes two or more arguments, with
2. Default values for one or more of those arguments.

The new function g is a specialization of f, with hard-coded values for some f's parameters.
Once defined, we can then call g with those arguments that have not yet been hard coded
Ex:

```
product_of x y z = x * y * z product_5 = product_of 5 product_5 = 2 3
```

Currying:
Transforms a function of multiple arguments to a series of functions of a single argument

When you "curry" a function, you convert a function that takes multiple arguments, e.g.:

```
function f(x, y) { return x + y; }
```

into a series of nested functions which each take a single argument:

```
function f(x){ function g(y) { return x + y } return g; }
```

```
- our curried function now takes a single argument
```

function f(x, y, z) { return x + y + z; } function f(x) { function g(y) { function h(z) { return x + y + z } return h; } return g; }

```
- In function h(z), the x and y values are captured which ultimately results in the inner-most function doing the original computation

Every function in haskell can be represented in curried form
Currying: the concept that you can represent any function that takes multiple arguments by another that takes a single argument and returns a new function that takes the next argument
y = f(a, b, c) => y = ((f a) b) c)
Each function takes one argument and returns another as its result (except for the last function which returns the result)

Currying PseudoCode:
Curry(Function f)
e = The expression/body of function f
for each parameter p from right to left:
f_temp = define a new lambda function with
1. p as its only parameter
2. e as its (expression)
e = f_temp
return f_temp

Everytime you define a function of more than one parameter in Haskell
Haskell automatically and invisibly curries it for you
Each successive function call adds its parameter to a closure!

TypeSignatures have syntactic Sugar for this:
Double -> Double -> Double -> Double
Double -> (Double -> (Double -> Double))

Currying enables partial function application
Since every haskell function is curried by default:
```

mult x y z = xyz and mult3 is \x -> (\y -> (\z -> (xyz)))

```
Thus when we call a function with less than the full number of arguments, the curried function will return closures that incorporate each value

Algebraic Data Types & Immutable Data Structures
Algebraic Data Type = describes a user-defined data type that can have multiple fields
The closest thing to algebraic data types would be C++ structs and enumerated types
We use algebraic data types to create complex data structures like trees.
```

data Drink = Water | Coke | Sprite | Redbull data Veggie = Broccoli | Lettuce | Tomato data Protein = Eggs | Beef | Chicken | Beans data Meal = Breakfast Drink Protein | Lunch Drink Protein Veggie | Dinner Drink Protein Protein Veggie | Fasting

```
Algebraic Data Type definitions in Haskell are like a combination of C++ structs and
```

```
C++ enumerated types

Each choice for a given type is called a "variant" or a "kind"

We can also define a more complex AFT where each variant also has one or more fields
(like a C++ struct)

In functional languages, we use pattern matching to process algebraic variables.


Using Algebraic Data Types to create a tree
Creating a binary search tree
We'll start by looking at the definition of a tree node ADT
```

```
data Tree = Nil | Node String Tree Tree
```

```
1st tree is the left child of the node
2nd tree is the right child of the node

Searching a Tree
```

```
data Tree = Nil | Node String Tree Tree search Nil val = False search (Node curval
left right) val | val == curval = True | val < curval = search left val | val > curval
= search right val
```

```
1st case is when curval and val are equal. Thus we can return True since we've found
the value
The second case is where if the (val) is less than the value in curval in the current
node, thus we will search left.
Third case is where if val is greater than value in curval in the current node, then
we will search right

To modify an existing tree by adding a new node to a tree:
add only part of the tree with the new node as well as its ancestors but keep all
other unaffected parts of the tree connected to the new part of the tree and then
forget the old tree
```

```
insert Nil v = Node v Nil Nil -- case where the root is empty insert (Node val left
right) v | v == val = Node val left right | v < val = Node val (insert left v) right |
v > val = Node val left (insert right v)
```

``` What's the Big 0? To add a new node, we just need to generate replacement nodes for the nodes between our new node and the root!

If a tree is balanced with n nodes, we need to create just logn new nodes

- this is because the height of the tree can be at most logn Virtually all of the nodes can be reused from the new tree

What happens to Old Nodes? This is where Garbage Collection comes in! Garbage collection is a language feature that automatically reclaims unused variables All functional languages have built-in Garbage Collection

Hash Tables are not efficient in Functional Languages as you would still need to regenerate the whole array if you wanted to insert, but searches are still O(1)
```

Linked Lists are efficient kinda. Addng to the front of the list is O(1), but adding somewhere in the middle will be O(N)

Many languages now provide immutable data structures from functional programming

- It helps simplify code, reduce bugs, and ease multi-processing